DiskTrie: An Efficient Data Structure Using Flash Memory for Mobile Devices

N. M. Mosharaf Kabir Chowdhury

Md. Mostofa Akbar

M. Kaykobad

Outline

- Problem statement
- Current status and motivation for a new solution
- Preliminaries
- DiskTrie Idea
- Results
- Limitations
- Future directions

Problem Statement

- Let *S* be a static set of *n* unique finite strings with the following operations:
  - Lookup (str) – check if the string str belong to the set *S*
  - Prefix-Matching (P) – find all the elements in *S* that have the same prefix P
- The Problem: An efficient data structure that can operate in low-spec mobile devices and supports this definition

Current Status

- At present, use of mobile devices and different sensor networks is increasing rapidly
- Mobile devices and embedded systems are characterized by –
  - Low processing power
  - Low memory (both internal and external)
  - Low power consumption
- Data structures and algorithms addressing these devices has huge application

Motivation for a New Solution

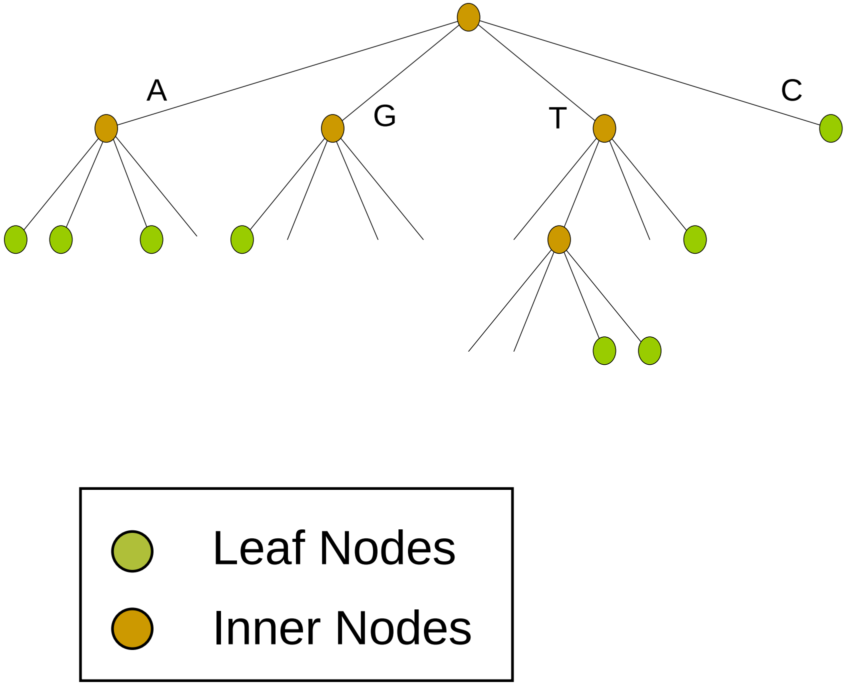
- Use of external memory is necessary
- Popular external memory data structures for computer include String B-tree, Hierarchy of indexes etc.
- The problem is still not very well discussed in case of flash memory (Gal and Toledo)
- Looking for a more space-efficient (both internal and external) data structure that is still competitive in terms of time efficiency

Flash Memory

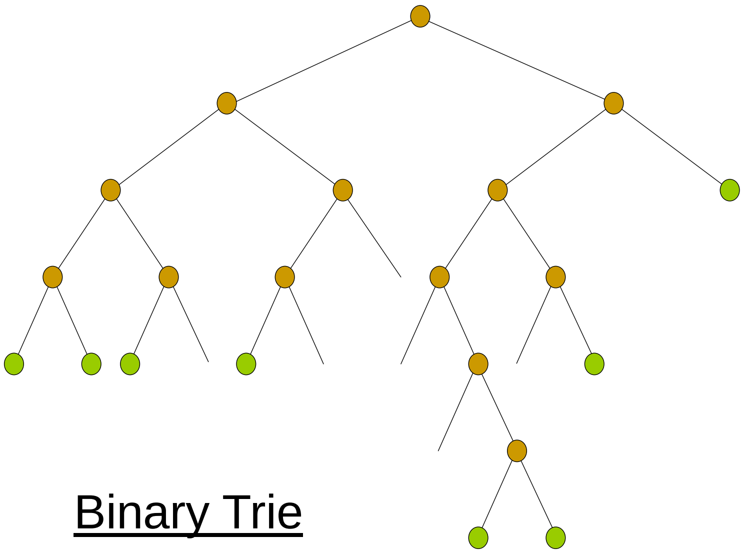
- Common memory that is extensively used in mobile/handheld devices
- Unique read/write/erase behavior than other programmable memories
- NOR flash memory supports random access and provides byte level addressing
- NAND flash memory is faster and provides block level access

Trie

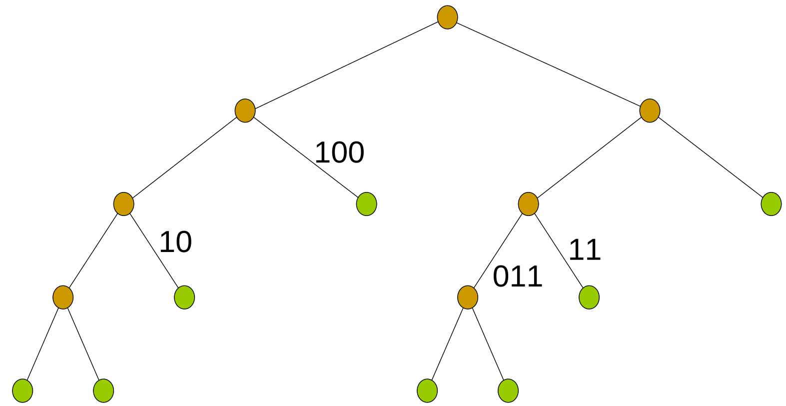
- A trivial trie is an *m-ary* tree
- Keys are stored in the leaf level; each unique path from the root to a leaf corresponds to a unique key
- Its search time can be considered as *O(1)*



Binary Trie and Path Compression



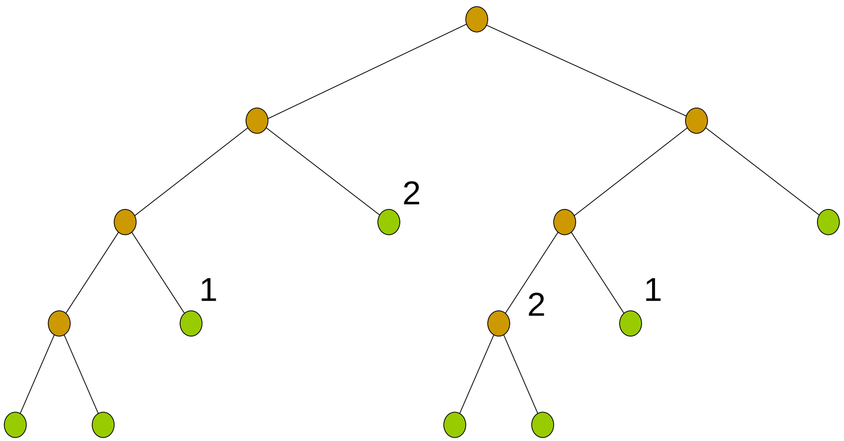
Binary Trie



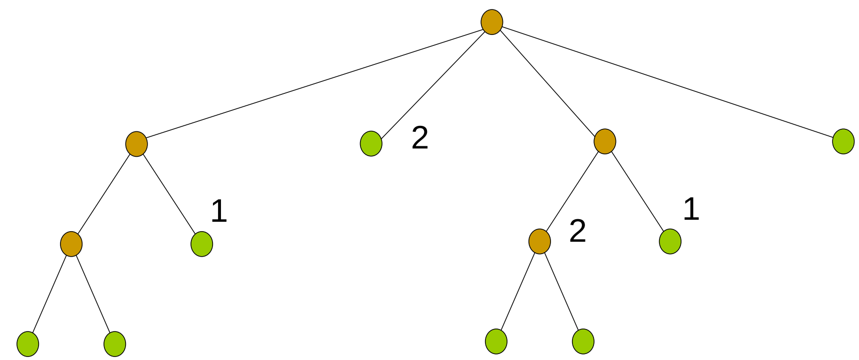
Path-compressed Binary Trie

- Binary encoding ensures every node to have a maximum degree of two
- Depth of the trie increases
- Path-compression is used to reduce this

Patricia Trie & LPC Trie



Patricia Trie



Level and Path-compressed Trie

- Patricia trie is similar to path-compressed one but needs less memory
- Finally, level and path-compressed trie reduces the depth but the trie itself does not remain binary anymore
- Nilsson and Tikkanen has shown that an LPC trie has expected average depth of $\Theta(\log^* n)$

DiskTrie Idea

- Static external memory implementation of the LPC-trie
- Pre-build the trie in a computer and then transfer it to flash memory
- Three distinct phases –
  - Creation in computer
  - Placement in flash memory
  - Retrieval

Creation and Placement

- All the strings are lexicographically sorted and placed contiguously in flash memory
- Nodes of the DiskTrie are placed separately from the strings and leaf nodes contain pointers to actual strings they represent
- Page boundaries are always maintained in case of NAND memory
- All the child nodes of a parent node are placed in sequence to reduce the number of pointers

Retrieval

- Deals with two types of operations:
  - Lookup
  - Prefix-Matching
- Lookup starts from the root and proceeds until the search string is exhausted
- Each time a single node is retrieved from the disk in case of NOR flash memory and a whole block for NAND type

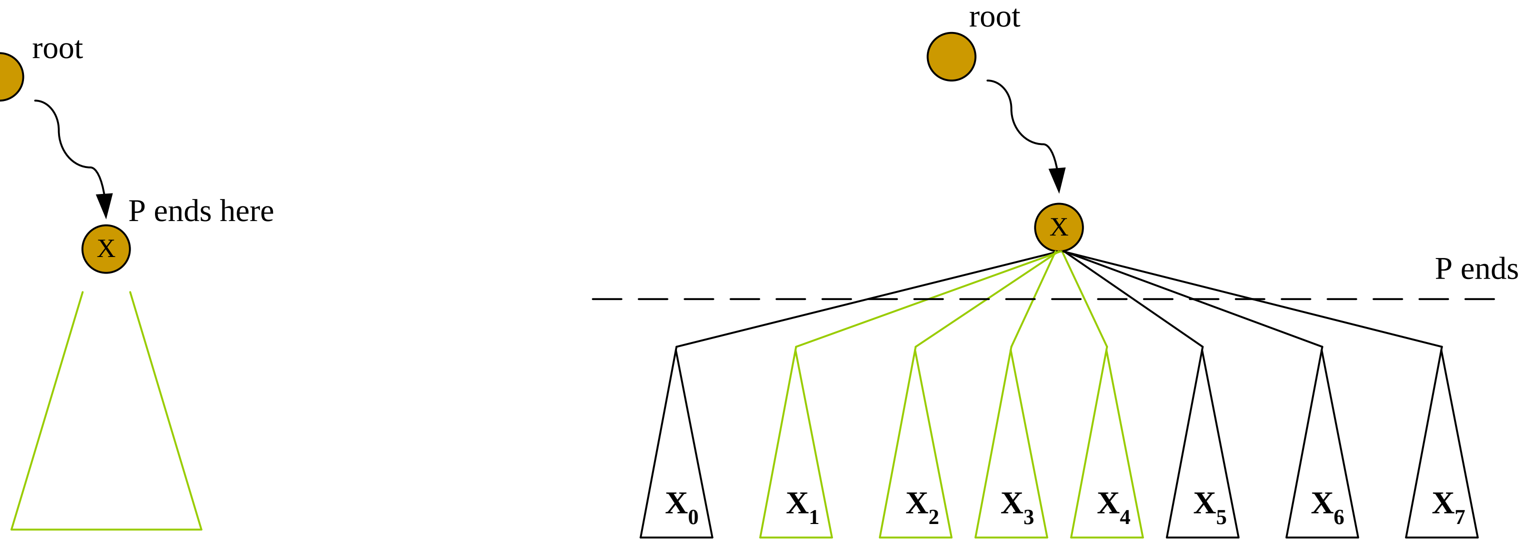
Lookup Algorithm

```
procedure Lookup (str)
{
-   currentNode ← root
-   while ( str is not exhausted & currentNode is NOT a
    leaf node)
    -   Select childNode using str
    -   currentNode ← childNode
-   end while

-   if ( error )
    -   return false
-   end if

-   return CompareStrings (str, currentNode→str)
}
```

Retrieval (Cont.)

- For Prefix-Matching operation, the searching takes place in two phases:
  - Identification of a prospective leaf node to find the *longest common prefix*
  - Identification of the sub-trie or tries that contain the results

Illustration of the Prefix-Matching Operation



(a) ‘P’ ends in a node

(b) ‘P’ ends in an arc

Prefix-Matching Algorithm

```
procedure Prefix-Matching (P)
{
-   currentNode ← root
-   while ( P is not exhausted & currentNode is NOT a leaf
    node)
    -   Select childNode using str
    -   currentNode ← childNode
-   end while

-   if ( error )
    -   return NULL
-   end if

-   lNode ← left-most node in the probable region
-   rNode ← right-most node in the probable region

-   return all strings in the range
}
```

Results
- Storage Requirement

- DiskTrie needs *two* sets of components to be stored in the external memory:
  - Actual Strings, and
  - The data structure itself
- Linear storage space to store all the key strings
- A Patricia trie holding n strings has $(2n – 1)$ nodes
- Hence, storage requirement for the total data structure is also linear
- While storing the nodes, block boundaries must be maintained. It results in some wastage

Results (Cont.)
- Complexity of the Operations

- **Lookup**
  - Fetch only those nodes from the disk that are on the path to the goal node
  - The number of disk accesses is bounded by the depth of the trie, which is in turn *Θ(log*n).*
    - *log*n* is the *iterative logarithm* function and defined as,
      - log*1 = 0
      - log*n = 1 + log*(ceil (log n)); for n > 1
  - Minimal internal memory required

Results (Cont.)

- **Prefix-Matching**
  - Probable *range* of the strings starting with the same prefix is identified using methods similar to Lookup. It takes *Θ(log*n)* disk accesses
  - In case of a successful search, it takes O(n/B) more disk accesses to retrieve the resultant strings if NAND memory is used (*B* is the block read size)
  - Sorted placement of the strings saves a lot of string comparisons
  - Internal memory requirement is minimal

Limitations

- Wastage of space in each disk block while storing the DiskTrie nodes
- In some cases, same disk blocks are accessed more than once

Future Directions

- More efficient storage management, specially removing the inherent wastage to maintain boundary property

- Take advantage of spatial locality

Thank You All !!!